**SOLID Principles**

### **Single Responsibility Principle:**

The Single Responsibility Principle states that a class should have only one reason to change, meaning it should have only one responsibility.

**Example:**

The UserManager class handles creating users and sending emails within the same class. This violates the Single Responsibility Principle because the class has different responsibilities in the same class.

Before SRP:

| public class **UserManager** {  public void **CreateUser**() {  // code to create a user  }    public void **SendEmail**() {  // code to send an email  }  } |
| --- |

After applying SRP, we split the responsibilities into separate classes:

| public class **UserManager** {  public void **CreateUser**() {  // code to create a user  }  }  public class **EmailSender** {  public void **SendEmail**() {  // code to send an email  }  } |
| --- |

Single Responsibility Principle makes the code more modular and maintainable.

### **Open/Close Principle:**

The Open/Closed Principle states that classes should be open for extension but closed for modification, meaning you should be able to extend their behavior without modifying their source code.

**Example:**

Consider a payment processing system that handles various payment methods such as credit card, PayPal, and cryptocurrency. Initially, the system only supports credit card payments, but later, you need to add support for PayPal payments without modifying the existing payment processing code.

Before applying OCP:

| public class **PaymentProcessor** {  public void **ProcessCreditCardPayment**(CreditCardPayment payment) {  // Code to process credit card payment  }  } |
| --- |

In the above code, the PaymentProcessor class directly handles credit card payments. If you want to add support for PayPal payments, you'll need to modify this class, violating the OCP.

After applying OCP:

| public interface **IPayment** {  void **ProcessPayment**();  }  public class **CreditCardPayment** : **IPayment** {  public void **ProcessPayment**() {  // Code to process credit card payment  }  }  public class **PayPalPayment** : **IPayment** {  public void **ProcessPayment**() {  // Code to process PayPal payment  }  }  public class **PaymentProcessor** {  public void **ProcessPayment**(IPayment payment) {  payment.ProcessPayment();  }  } |
| --- |

In this refactored code:

We introduce an interface IPayment representing the behavior of a payment method. CreditCardPayment and PayPalPayment are concrete implementations of IPayment, each responsible for processing their respective payment methods. PaymentProcessor now accepts any object that implements IPayment. It's open for extension (you can add new payment methods) but closed for modification (existing code doesn't need to change when adding new payment methods).

By adhering to the Open/Closed Principle, we've made our code more flexible and maintainable. We can easily add support for new payment methods without modifying existing code, promoting separation of concerns and reducing the risk of introducing bugs.

### **Liskov Substitution Principle:**

It’s an extended version of the Open/Close Principle. The Liskov Substitution Principle states that the object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class. That means the child class objects should be able to replace parent class objects without changing the correctness or behavior of the program.

**Example:**  
  
Before applying LSP:

| public class **Employee** {  public virtual decimal **CalculateSalary()** {  return **0**;  }  }  public class **FullTimeEmployee** : **Employee** {  public override decimal **CalculateSalary**() {  // Calculation based on full-time employment  return **5000**;  }  }  public class **Contractor** : **Employee** {  public override decimal **CalculateSalary**() {  // Calculation based on contract rate and hours worked  return HourlyRate \* HoursWorked;  }  public decimal HourlyRate { get; set; }  public decimal HoursWorked { get; set; }  } |
| --- |

In this example, both FullTimeEmployee and Contractor inherit from the Employee class. However, the way salary is calculated differs significantly – full-time employees have a fixed salary, while contractors' salaries depend on their hourly rate and hours worked. This violates the Liskov Substitution Principle because substituting an Employee with its subclasses may lead to unexpected behavior due to differing implementations of the CalculateSalary method.

After applying LSP:

| public interface **IEmployee** {  decimal **CalculateSalary**();  }  public class **FullTimeEmployee** : **IEmployee** {  public decimal **CalculateSalary**() {  // Calculation based on full-time employment  return **5000**;  }  }  public class **Contractor** : **IEmployee** {  public decimal HourlyRate { get; set; }  public decimal HoursWorked { get; set; }  public decimal **CalculateSalary**() {  // Calculation based on contract rate and hours worked  return HourlyRate \* HoursWorked;  }  } |
| --- |

In the refactored code:

We introduce an interface IEmployee representing the behavior of any type of employee. Both FullTimeEmployee and Contractor implement the IEmployee interface, ensuring they adhere to the Liskov Substitution Principle by providing the expected behavior without changing the base class behavior. This design allows for more flexibility, as both full-time employees and contractors can be treated interchangeably wherever employee-related operations are expected.

In real life, adhering to the Liskov Substitution Principle ensures that software components can be easily extended and replaced with compatible alternatives without introducing unexpected behavior, promoting code reuse and maintainability.

### **Interface Segregation Principle:**

The Interface Segregation Principle states that a client should not be forced to implement an interface that it doesn't use. Instead of creating large, monolithic interfaces, it's better to segregate them into smaller, specific interfaces.

**Example:**

Before applying ISP:

| public interface **IWorker** {  void **Work**();  void **Eat**();  void **Sleep**();  } |
| --- |

In this interface, all workers are forced to implement methods for working, eating, and sleeping. However, not all workers might need to implement all these methods. For example, a robot worker might not need to eat or sleep.

After applying ISP:

| public interface **IWorker** {  void **Work**();  }  public interface **IEater** {  void **Eat**();  }  public interface **ISleeper** {  void **Sleep**();  } |
| --- |

In the refactored code, we have segregated the monolithic interface into smaller interfaces, each representing a specific behavior. Now, classes can implement only the interfaces they need, adhering to the Interface Segregation Principle.

By adhering to the Interface Segregation Principle, we ensure that our interfaces are more focused, leading to better maintainability and avoiding unnecessary dependencies.

### **Dependency Inversion Principle (DIP):**

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. This principle promotes loose coupling between classes and makes the code more flexible and easier to maintain.

Example:

Before applying DIP:

Suppose we have a web application that interacts with a database to retrieve user information.

| public class **UserRepository**  {  public User **GetUser**(int userId)  {  // Code to fetch user from the database  }  }  public class **UserController**  {  private readonly UserRepository \_userRepository;  public **UserController**()  {  \_userRepository = new UserRepository();  }  public User **GetUserDetails**(int userId)  {  return \_userRepository.GetUser(userId);  }  } |
| --- |

In this code, UserController directly depends on UserRepository, creating a tight coupling between the two classes.

After applying DIP:

We'll introduce an abstraction layer (IUserRepository) between the high-level module (UserController) and the low-level module (UserRepository).

| public interface **IUserRepository**  {  User **GetUser**(int userId);  }  public class **UserRepository** : **IUserRepository**  {  public User **GetUser**(int userId)  {  // Code to fetch user from the database  }  }  public class **UserController**  {  private readonly IUserRepository \_userRepository;  public **UserController**(IUserRepository userRepository)  {  \_userRepository = userRepository;  }  public User **GetUserDetails**(int userId)  {  return \_userRepository.GetUser(userId);  }  } |
| --- |

In this refactored code:

* UserController depends on the IUserRepository interface instead of the concrete UserRepository class.
* UserRepository implements the IUserRepository interface.
* Now, UserController is decoupled from the specific implementation of the data access layer. It only depends on the abstraction (IUserRepository), adhering to the Dependency Inversion Principle.

This design allows for flexibility and easy swapping of different data access implementations without modifying the UserController class, promoting code maintainability and testability.

https://www.adaface.com/blog/solid-principles-interview-questions/